Evolution of Corridor Following Behavior in a Noisy World

Craig W. Reynolds
Electronic Arts
1450 Fashion Island Boulevard
San Mateo, CA 94404, USA
telephone: 415-513-7442 / fax: 415-571-1893
creynolds@ea.com
cwr@red.com

Abstract

Robust behavioral control programs for a simulated 2d vehicle can be constructed by artificial evolution. Corridor following serves here as an example of a behavior to be obtained through evolution. A controller's fitness is judged by its ability to steer its vehicle along a collision free path through a simple corridor environment. The controller's inputs are noisy range sensors and its output is a noisy steering mechanism. Evolution determines the quantity and placement of sensors. Noise in fitness tests discourages brittle strategies and leads to the evolution of robust, noise-tolerant controllers. Genetic Programming is used to model evolution, the controllers are represented as deterministic computer programs.

1 Introduction

Designing reactive controllers for autonomous agents can be a challenging task. For increasingly complex behavior, building controllers by hand becomes prohibitively difficult. As suggested in [Cliff 1993b], a promising alternative is to *evolve* the controllers, using their ability to perform the desired behavior as a measure of *fitness*. In these experiments, *corridor following* behavior is used as a simple test case, representative of the more complex behaviors to which this approach might eventually be applied.

Previous work [Reynolds 1993b] has shown that the Genetic Programming Paradigm [Koza 1992] can be used to automatically create control programs which enable a simple moving 2d vehicle to avoid collisions with obstacles by mapping sensory input (range data) into motor output (steering action). In those experiments all fitness tests were identical. As a result, the control programs evolved to use *brittle* strategies. Their success was like a "house of cards" which stands only until anything changes.

In the absence of variability in fitness testing, evolution will discover solutions that capitalize on the deterministic, precisely repeatable nature of the fitness tests. Evolved controllers will come to depend on utterly insignificant coincidental properties of the vehicle's sensors, its actuators, and their interaction with the environment.

The current work concerns an approach to avoiding this brittle behavior and seeks to evolve robust, general purpose control programs for the corridor following problem. Determinism is removed from fitness testing by injecting noise into the system. This noise will tend to "jiggle" coincidental relationships between elements of the system and so tend to discourage evolution from capitalizing on them. A house of cards cannot be built on a shaky table.

The controllers evolved in this work are computer programs composed of basic arithmetic operations, conditionals, and a function to aim and read a nonlinear proximity sensor. The number and orientation of sensors are determined through evolution of control programs. On each simulation step control programs read their sensors and compute a steering angle. The vehicle always moves forward at a constant rate, so steering is its only means of avoiding collision.

These experiments have produced robust control programs capable of corridor following behavior in the presence of noise. Figure 1 shows some examples of successful behavior.

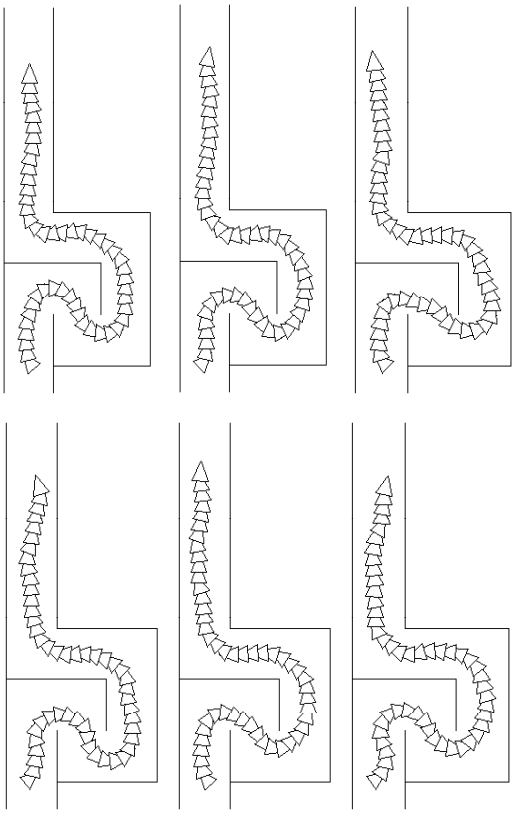

Figure 1: Several collision-free runs.

To appear in Simulation of Adaptive Behavior 1994.

2 Related work

An early series of experiments [Reynolds 1993b] used Genetic Programming to create reactive controllers for a similar obstacle avoidance task. The fitness test employed a single, precisely repeatable simulation-based fitness test. This allowed evolution to take the easy path to discover a program which only solved this one specific control task. There was no incentive, no survival advantage, to find a controller that could generalize. As a result, the evolved controllers were "brittle" and could not solve similar but slightly different problems.

Subsequent experiments [Reynolds 1994a] attempted to use noise to promote robust solutions to the corridor following task, but were unsuccessful. The most significant difference between those experiments and these is the addition of a syntactic constraint to the sensor-reading function. In the earlier work, the control program could rotate its sensors relative to the vehicle. In the current work, the sensors are fixed to the vehicle during its run. Sensor orientation is still subject to change by the action of evolution.

Closely related to these experiments is the work of the Evolutionary Robotics Group at the University of Sussex. While using a different model of evolution (SAGA [Harvey 1992]) and a different model of controller architecture (dynamic, recurrent neural nets [Cliff 1993a], [Cliff 1993b], and [Harvey 1993]) they have investigated closely related problems in evolution of robotic controllers. They were the first to document the beneficial role of noise in the evolution of robust robotic controllers. The general approach used here, of evolving stimulus-response behavior based on simulated performance using simulated perception inside a closed simulated world, was originally inspired by [Cliff 1991a] and [Cliff 1991b]. The specific technique used here, of taking the "worst of four noisy trials" came directly from [Harvey 1993] and [Cliff 1993a].

The experiments reported here have much in common with some of Koza's work, particularly the evolution of behaviors such as "wall following" and "box pushing" as reported in [Koza 1992]. See also Simon Handley's GP robotics work [Handley 1994].

The evolution of robust controllers is related to the larger problem of generalization in evolutionary computation. This issue of generalization is an active area of research in many branches of evolutionary computation. In GP, see for example [Tackett 1994] on the evolution of generality in a classification problem, and [Kinnear 1993] on generalization in sorting.

Earlier work on obstacle avoidance behavior based on remote (distal) sensors (as opposed to touch sensors) for vehicles moving at "moderate speed" can be found in [Reynolds 1987], [Reynolds 1988], [Mataric 1993] and [Zapata 1993], among many others.

A classic reference on controllers and behaviors for this class of *vehicle* is [Braitenburg 1984] which is highly recommended.

3 Vehicle

The design of the simulated vehicle used in these experiments is kept intentionally vague and abstract. It could equally well represent an animal as a wheeled or legged robotic vehicle. The intent is to gloss over the low level details of locomotion and to concentrate instead on the more abstract issues of "steering" and "path determination." (Not "path *planning*," since these reactive controllers neither plan nor learn.) In order to survive, the controllers need only steer along a clear pathway while avoiding contact with the danger region surrounding it. The skill involved is similar to that required by a squirrel running along a tree branch, or by an automobile's driver, negotiating through a narrow alley.

The control programs evolved by Genetic Programming represent the vehicle's "thought process" for a single simulation step. During fitness testing, the evolved program is run at each time step of the simulation. Using its sensors, the program inspects the environment surrounding the vehicle from its own point of view (that is, relative to the vehicle's local coordinate space), performs some arithmetic and logical processing, and decides how to steer (adjust the heading of) the vehicle. The value returned by the control program is interpreted as a steering angle. The vehicle then automatically moves forward by a constant amount (half of its body length). The fitness test continues until the vehicle takes the required number of steps (50), or until it collides with one of the obstacles. The raw fitness score for each corridor run is the number of steps taken divided by the maximum number of steps, producing a normalized score between 0 and 1, with 1 being best.

These vehicles have a fixed minimum turning radius because the maximum per-simulation-step turn is limited to ±0.05 revolutions (18 degrees or 0.31 radians). This limitation implies a minimum turning circle which is somewhat larger than the width of the corridors of the obstacle course. As a result, the vehicle cannot spin in place, it cannot turn around in the corridor, and its only choice is to travel along the corridor.

Limiting turning radius produces a model of a vehicle moving at "moderate" speed. This qualitative description is intended to capture a relationship between the vehicle's momentum and its available turning acceleration. At "low" speed a vehicle has relatively little momentum, in a single time step it might be able to bring itself to a stop, or make an abrupt change of heading. At moderate speed momentum begins to dominate acceleration and changes of heading require many time steps. In this speed regime, maneuvers are less abrupt and paths tend to curve more gently, producing a motion more like running than crawling.

4 Corridor and Fitness Testing

The training environment used in these experiments was designed to test a control program's ability to follow a corridor, and to quickly reject those which are completely unsuited to the task. This approach allows the majority of the computational effort to go into testing higher fitness individuals. A fitness function based on this kind of simulation has the desirable property that execution time is roughly proportional to the individual's fitness.

Figure 2 shows the corridor and an series of increasingly successful runs. The vehicle is initialized in the center of

the corridor and pointing toward one wall or the other. Initial random headings range from 0.1 to 0.15 revolutions (36 to 54 degrees) off of the corridor's midline. If a controller turns continuously (see Figure 2(a)) or does not steer at all (see Figure 2(b)), it will run into a wall almost immediately. To survive more than a few steps the controller must develop the skill of sensing and turning away from a glancing collision so it can proceed down the corridor. To reach higher fitness levels the vehicle must be able to safely negotiate U-turns in both directions, right-angle turns in both directions, and long straight corridors.

In the early stages of evolution most of the controllers are quite inept and so incapable of following the corridor for long. During this stage it is important to notice and encourage progress, no matter how slight. In the initial population only a few controllers can take more than one or two collision-free steps so the scoring system must differentiate between degrees of ineptness, as illustrated in Figure 2. Later in the run some controllers will have progressed to the point where they can occasionally make a collision-free run through the corridor. At this stage the point of fitness testing begins to be reliability. One way to address this is to look at the controller's performance on a series of corridor runs. These considerations lead to the following scheme for each fitness trial: the controller is allowed to make a run through the corridor, if no collisions occur it is allowed to make another run, and so on up to a maximum of 16 runs. This creates a selection pressure for controllers to be able, at least occasionally, to make it all the way through the corridor so as to have a chance at another run. This approach also serves to further focus effort on promising controllers: we don't even bother with a second trial unless the controller has "proven" itself worth on the first trial.

The scores of all runs of a trial (up until the first collision) are weighted and summed. The first collision-free run is worth 1/2, the second is 1/4, and so on. Figure 3 shows the cumulative fitness assigned to a controller based on the number of collision-free runs. This weighting scheme captures the idea of an open-ended reliability rating, but is probably not significant now that tournament selections is being used and all that matters is fitness rank. Monotone functions do not alter rank.

An obstacle course used in earlier experiments had a longer straight-away before and after the first turn. This arrangement seemed to promote premature convergence, particularly in small populations. Some individuals would discover how to make it to the first turn (and sometimes past it) but they would then so dominate the population that diversity would be lost and the population would never discover how to get around the second turn. In the subsequent experiments described here these problems were addressed in two ways. First, the long straight-aways were moved from the beginning of the course to the end. This

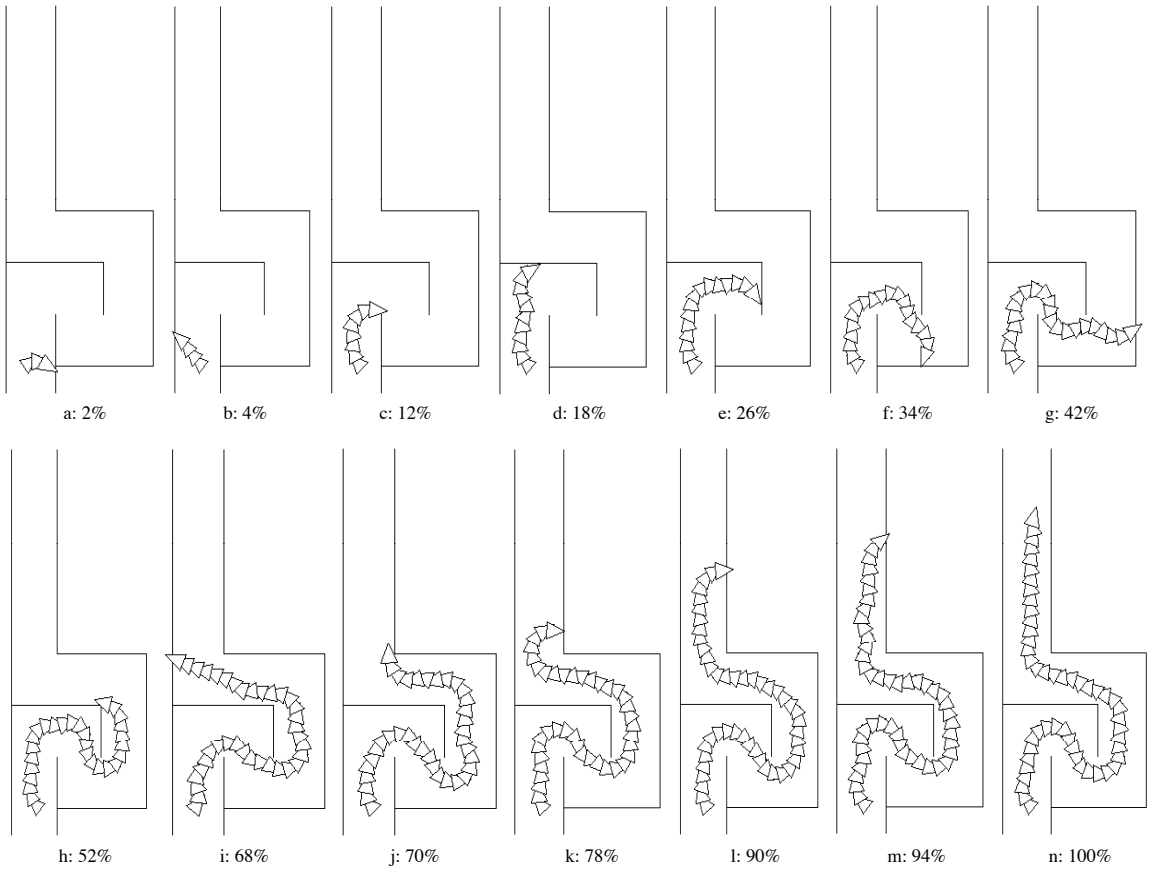

Figure 2: An assortment of runs, arranged in order of increasing scores. All but the last (*n*) end prematurely because of a collision.

forced the turning problem to be addressed sooner, and weeded out the non-turners sooner. Second, the entire obstacle course was mirrored at random about the axis of the first corridor. This procedure ensured that evolving controllers could turn equally well in both directions and so prevented convergence towards right-turners.

Because the training corridor is laid out on a grid, it has some built-in regularities. The passageway has uniform width throughout. All walls meet at right angle corners. Presumably evolution will construct controllers which depend on these regularities. That is, in the absence of counter-examples, it will assume that all corridors are the same width. A different kind of training environment may be required to evolve controllers capable of following irregular corridors.

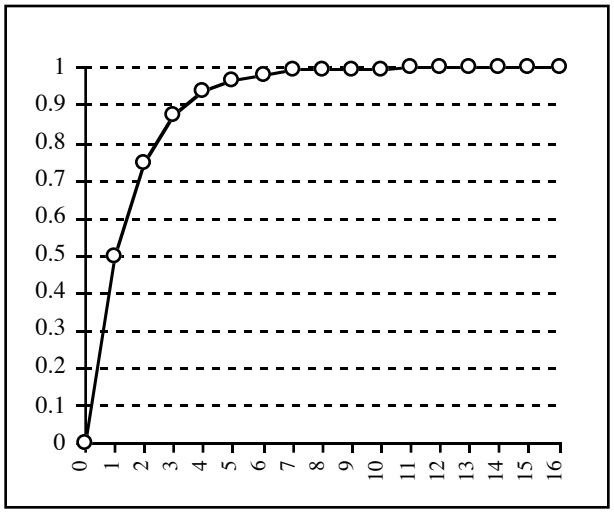

Figure 3: Fitness versus number of sequential collision-free runs through the corridor.

5 Genetic Programming Considerations

The technique used to evolve computer programs in this work is known as Genetic Programming and was invented by John Koza. The best reference on this technique and its application is [Koza 1992]. While often used as a generational technique, it is also possible to combine Genetic Programming with Steady State Genetic Algorithms [Syswerda 1989], [Syswerda 1991] as described in [Reynolds 1993a].

A very brief description of Steady State Genetic Programming (SSGP) follows. First a population of random programs is created and fitness tested. (In these experiments the population consisted of 2000 to 10000 programs.) Thereafter SSGP proceeds by: (1) choosing two *parent* programs from the population, (2) creating a new *offspring* program from them, (3) fitness testing the new program as described in the previous section, (4) choosing a program to remove from the population to make room, and (5) adding the new program into the population. The parent programs are chosen in a way that favors the more fit while not totally ignoring the less fit, thus balancing *exploration* of the whole gene pool with *exploitation* of the champions. In these experiments this choosing is done using *tournament selection* with k=7, that is: seven individuals are chosen from the population at random, and the most fit of those seven is selected as the winner. The recombination of two parents to form a new offspring is accomplished by the Genetic Programming crossover operator. GP crossover is a little like "random cut and paste" but is done in a why that guarantees the new program's syntactic correctness.

Selecting a program to remove from the population could be done by using inverse tournament selection: taking the least fit of seven randomly chosen programs. However the greedy nature of SSGP, combined with the noisy fitness testing used in these experiments, leads to the possibility of a mediocre-but-lucky program receiving an undeservedly high fitness and going on to dominate the population. To combat this possibility a modified removal policy was used in these experiments: half the time inverse tournament selection was used, the other half of the time an individual was selected for removal at random (without regard to fitness). Hence all programs, even the best one, had a certain small but non-zero possibility of being removed at each SSGP step. This tended to ensure that the population could not stagnate with a collection of mediocre-but-lucky programs, winning strategies were continually being retested.

Because steady state genetic computation proceeds individual by individual, there is no demarcation of generations. However it is often convenient to describe the progress or length of a SSGP run in terms of "generation equivalents:" processing as many new individuals as there are programs in the population.

Applying Genetic Programming to a problem requires specifying several parameters such as the genetic population size and the fitness function (both described above). In addition we must specify the *functions* and *terminals* that define the language in which evolved program will be expressed. In these experiments the terminals were simply random floating point numbers. The function set consisted of:

```
+
-
*
%
abs
iflte
look-for-obstacle
```

The first three are the standard Common Lisp function for addition, subtraction, and multiplication. The `%` and `iflte` are standard GP functions [Koza 1992]: `%` is "protected divide" (returns zero when denominator is zero), and `iflte` is an arithmetic conditional (if A is less than or equal to B, then C, else D). `abs` is the standard Common Lisp function for absolute value.

The `look-for-obstacle` function is specific to the obstacle avoidance problem. It takes a single numeric argument which represents an angle relative to the current vehicle heading. Angles are specified in units of *revolutions*, a normalized angle measure: 1 revolution equals 360 degrees or 2π radians. `look-for-obstacle` points its sensor in the given direction and returns a measure of obstacle proximity. In this implementation, the range is computed by performing a 2d ray-tracing operation on the obstacles.

The Genetic Programming substrate used here, and the application-specific functions for the simulation, were originally developed on Symbolics Lisp Machines. For the current series of experiments, the software was ported to Macintosh Common Lisp (version 2.0p2) and was run on Macintosh Quadra 950 workstations. In this implementation, an average fitness test (composed of up to 64 corridor runs) in run z6 took about 65 seconds to perform.

6 Results

Three runs will be discussed in this section. One uses fixed sensor positions, the other two allow the sensor placement to evolve. The genetic population size and upper limit on evolved program size also differ between the runs:

run:	population:	sensors:	program size limit:
z4	10000	fixed	50
z6	2000	variable	35
z7	10000	variable	35

Run "z4" used a fixed-sensor model. The vehicle was defined to have exactly nine sensors, spaced 1/16 of a revolution (22.5 degrees) apart across the vehicle's front. (If the vehicle's heading is "north," this would correspond to placing sensors at compass points: W, WNW, NW, NNW, N, NNE, NE, ENE, and E.) In Genetic Programming terms, this was implemented by adding a set of nine sensor-reading functions to the GP function set. For comparison with the other two runs described below, this fixed sensor model could also have been implemented by restricting the argument to `look-for-obstacle` to be one of the nine numeric values (3/4, 13/16, 7/8, 15/16, 0, 1/16, 1/8, 3/16, 1/4). In this model, the evolved control programs would call various sensor-reading functions, perform some arithmetic and logical processing, and return a steering angle. Evolution could not alter the sensor placement, it could only select which sensors to use, and how to combine their readings.

This formulation of the corridor following turned out to be very easy. So much so that the GP system almost solved it by random search. The initial generation of a GP run corresponds to random search over the space of programs (subject to the limitation of size, function set, and terminal set which are parameters to GP). During the initial generation of 10000 random programs for run "z4," the best program had a fitness of 76.5%. This value represents better performance than evolved in any of the previous variations of this problem as reported in [Reynolds 1994a], even though this program was found by random search before the beneficial effects of evolution were applied to the population. During the third generation equivalent (around individual number 35000) of run "z4" it began to attain best-of-population fitness scores above 99%.

Figure 5 summarizes the z4 population's fitness distribution over time. It shows a time series of discrete fitness histograms portrayed as a bilinear surface. The population's progress towards increasing fitness over time can be seen. Dominating this landscape are fin-shaped features probably caused by two artifacts of the fitness function. The corridor is a series of turns, most runs end at a turn, so scores tend to be clumped at certain values. Then as described in Figure 3, this distribution is repeated at half scale, at quarter scale, and so on for additional runs.

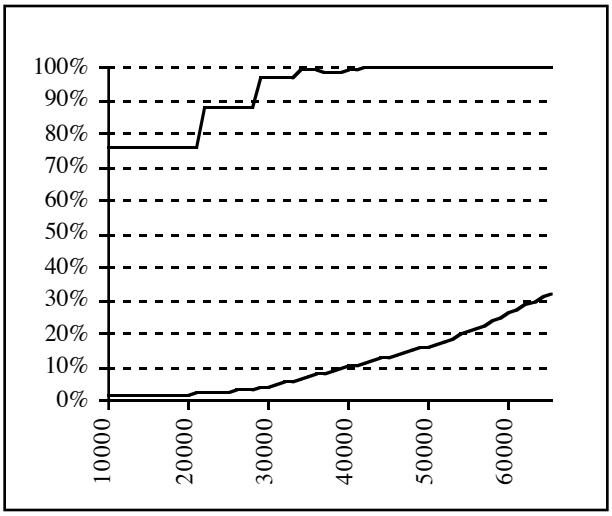

Figure 4: Fitness plots for run z4.
(best-of-population and population average)

Koza has described what he calls the "lens effect," [Koza 1994] the way in which every representation alters the difficulty of a given problem. Koza explores the lens effect by looking at the distribution of fitness values found during random search through the specified program space. In those terms, it appears that the fixed sensor representation of the corridor following problem is a "lens" that makes the problem quite easy to solve. An analysis of fitness distribution in the initial generation of these three runs can be found in [Reynolds 1994b]. By that metric it appears that the fixed sensor representation is easier than the variable sensor representation, which is in turn easier than the original "roving" sensor representation of [Reynolds 1994a].

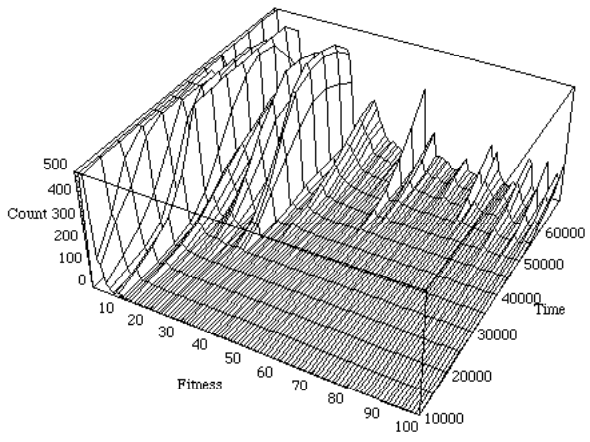

Figure 5: Fitness histograms of run z4 over time.

While not the best-of-run, one high scoring (99.82%) and compact (size 15) controller which caught the author's eye was this elegant three sensor design that appeared during the

fourth generation-equivalent (individual number 46515, see Figure 6). It is shown here hand-simplified down to size 7:

```
(% (- (obs-3/16) (obs-13/16))
   (- (obs-15/16) 2.5))
```

This program works by comparing the proximity of obstacles (walls) at relative headings of ±3/16 (the expression `(- (obs-3/16) (obs-13/16))` returns a value between -1 and 1 which indicates relative lateral proximity), then scales that value down by dividing it by a number ranging between -2.5 and -1.5 which is related to the obstacle proximity at a heading of -1/16. This appears to increase the rate of turn as to the amount of free space "ahead" decreases.

The controllers evolved in run z4 were both more and less complicated than the individual discussed above. Many were variations on the same theme: divide the relative lateral proximity by a sensor-dependent scalar, most varied only in the form of modulation. At the end of the run, after about 6.5 generation equivalents, there were two individuals who tested at 100% fitness. After simplification by hand, they were of size 5 and 33:

```
(% (- (obs-3/16)
      (obs-13/16))
   -1.995)

(% (- (obs-3/16)
      (obs-13/16))
   (- (abs (iflte (obs-15/16)
                  0.0
                  (+ (obs-3/16)
                     (obs-7/8)
                     (obs-15/16))
                  (* (* (iflte (obs-1/8)
                               (obs-0)
                               (obs-1/8)
                               0)
                         (iflte (obs-3/16)
                                (- (obs-1/8)
                                   (obs-3/16))
                                (obs-7/8)
                                0)
                         (obs-13/16)
                         0.5)
                      (obs-0)
                      (obs-1/8)
                      0.2)))
      2.0))
```

Based on those results, run z6 was designed to test whether evolution could correctly determine quantity and placement of sensors in addition to determining how to combine the various sensor readings into a correct steering angle. The approach was to restrict the argument to `look-for-obstacle` to be a number. That is, each call to `look-for-obstacle` has a constant numeric argument representing a certain angle. Any number is allowed, but more complicated expressions, particularly those involving additional calls to `look-for-obstacle` are not allowed. When a new program is created (randomly or by crossover) it is checked to ensure that arguments to `look-for-obstacle` are each numeric constants. When non-constant arguments are found, they are replaced by a random number between 0 and 1. As a result, each call such as `(look-for-obstacle 0.17)` corresponds to a sensor fixed at an arbitrary angular offset from the vehicle's heading. In this way, sensor *morphology* can evolve in parallel with the processing needed to map sensor data into a steering signal.

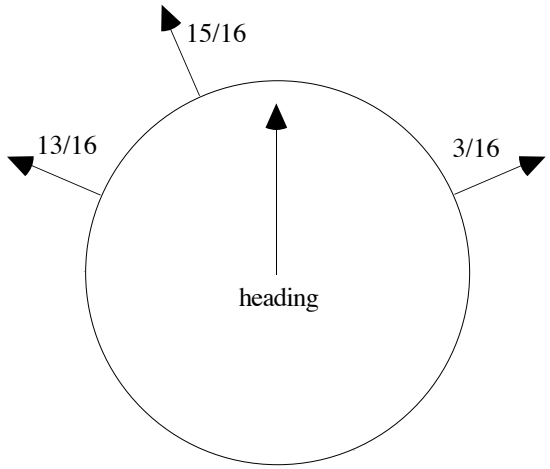

Figure 6: Sensor morphology of individual z4-46515.

Sensors pointing in certain directions (such as directly ahead) are presumably more relevant to corridor following than sensors pointing in other directions (such as directly behind). Therefore certain sensors, represented as code fragments, will tend to increase the fitness of programs in which they appear. This is known as the *constructional fitness* of the code fragment [Altenberg 1994]. Programs containing the more useful sensor-defining fragments will have a survival advantage, and so those fragments will tend to proliferate in the population. In this way the evolutionary process decides which sensor directions are most useful for solving the corridor following problem.

On the tenth generation equivalent of run z6, the best-of-population individual has a fitness of 98.03%. This run developed a rather elaborate *hitch-hiking intron*[1] that has spread throughout the population, so while the best-of-population individual is almost as large as it can be (size 34), most of its genetic material is irrelevant. After removing the irrelevant *intron wrapper* this size 10 program remains:

```
(* (look-for-obstacle 0.13)                    A
   (look-for-obstacle 2.0)                     B
   (- (look-for-obstacle 0.81)                 C
      (look-for-obstacle 0.17)))               D
```

While based on a slightly different approach, this program shares certain features of the program from run "z4" analyzed above. Specifically, the subtraction (on lines C and D) is computing the same sort of lateral proximity measure by taking the difference between a left pointing sensor

[1] By analogy with its usage in biological genetics, the term *intron* is used in GP to refer to code that is included in an evolved program but which does not contribute to the program's action or result. An intron is part of the genotype but not of the phenotype. When an intron becomes structurally associated with a highly fit code fragment, the intron can hitch-hike and so proliferate through the population. In the case of run z6, essentially all programs had a large no-op conditional wrapped around the active code.

and a right pointing sensor. In fact the left sensor used is essentially identical to the left sensor used in the "z4" program. The pair of sensors used here are not quite symmetrically placed: +0.17 revolutions on the right and -0.19 on the left. The lateral proximity value is then scaled down by multiplication by two numbers each of which are between 0 and 1. The sensor specified on line B points directly ahead (2.0 revolutions is the same angle as 0 revolutions) and the sensor on line A points 0.13 revolutions (47 degrees) to the right. Hence the lateral proximity signal is reduced as the obstacle-free path, either ahead or to the right, increases.

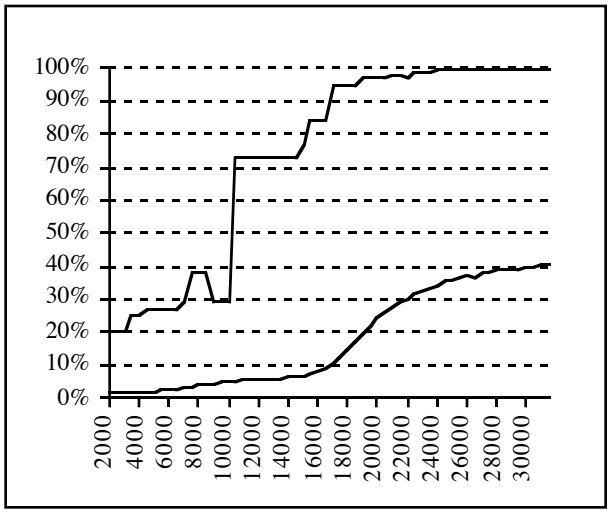

Figure 7: Fitness plots for run z6.
(best-of-population and population average)

Essentially all of the high-fitness individuals in run z6 use this same basic framework, within which there are a few variations of the sensor placement. For example the best-of-population individual at 21600 is identical except that it uses the values 0, 2.0, 0.82, and 0.17 (on lines A, B, C, and D, respectively). Effectively it has become a three sensor design, since both forward-pointing sensors are at the same angle. It is almost exactly symmetrical, which seems appropriate for its task.

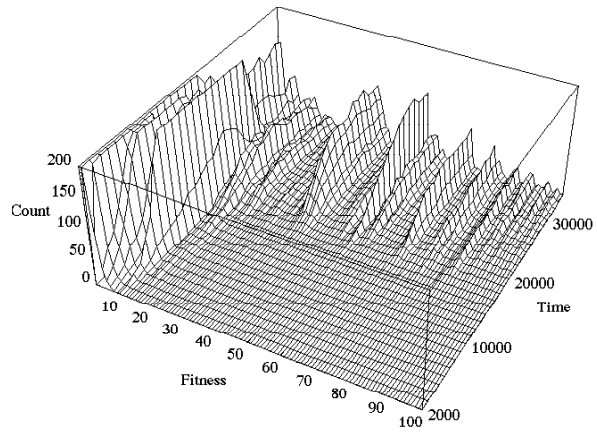

Figure 8: Fitness histograms of run z6 over time.

Run z6 found what might be regarded as a competent controller, but one that is clearly less than perfect. To attain 100% fitness in these experiments, a controller must be able to execute 64 (16 times 4) consecutive perfect runs through the corridor, for a total of 3200 correct steps. The best-of-generation fitness in z6 was always below 100%. This implies that the failure rate for these controllers is at least once in 64 runs. To put this in perspective, imagine a human driver who averages one collision every 64 automobile trips! While there is clearly room for improvement in the controllers from run z6, there is also evidence that the population became converged and would be unable to progress any further.

Another run was attempted to duplicate and hopefully surpass the results of z6. Run "z7" was identical to z6 except that the population was increased by a factor of five to 10000. Unfortunately the run appeared to "top out" running out of steam perhaps because the population converged too rapidly on an inferior strategy. See Figure 9. After about 8.5 generation equivalents, the best of population was stuck at 40% and population average was leveling off at 15%. This outcome demonstrates that evolutionary computation is a stochastic technique, and not all runs succeed. There are suggestions that most reliable way to use computational resources is to use a series of many smaller runs in place of a single large run.

7 Conclusions

These results indicate that behavioral control programs can be evolved using Genetic Programming and a noisy simulation-based fitness test. The solutions discovered by this process are simple and robust. It appears that noise in fitness testing discourages strategies that are brittle, opportunistic, or overly complicated. The only solutions that can survive noisy fitness testing are compact and robust.

8 Future Work

As corridor following behavior developed in these experiments it became clear that *reliability* is a difficult property to measure. While competent behavior clearly arose, it is hard to measure just how robust and how reliable these controllers are. If we wanted to measure the reliability of a human automobile driver, what would be the criterion? Would we count the number of collisions over a long period of time, say a year? How would we differentiate between all of the individuals who had zero collisions? Is a year a long enough test period, and if not, how long are we willing to wait for an answer? All of these same issue come up in trying to rate the reliability of the corridor following controllers evolved here. A topic of future study will be to determine the most efficient techniques for testing reliability.

The competent behavior evolved in these experiments represented a breakthrough after a long series of unsuccessful experiments in applying very similar GP techniques to the same problem. The fact that a seemingly tiny change in the GP representation (restricting the argument of `look-for-obstacle` to be a constant) made a huge difference in the difficulty of the problem is a tantalizing result which deserves further study. What does this say about the problem

domain? What does it say about Genetic Programming? How can we characterize this representational difficulty, and how can we avoid it in the future? Some first thoughts are explored in [Reynolds 1994b].

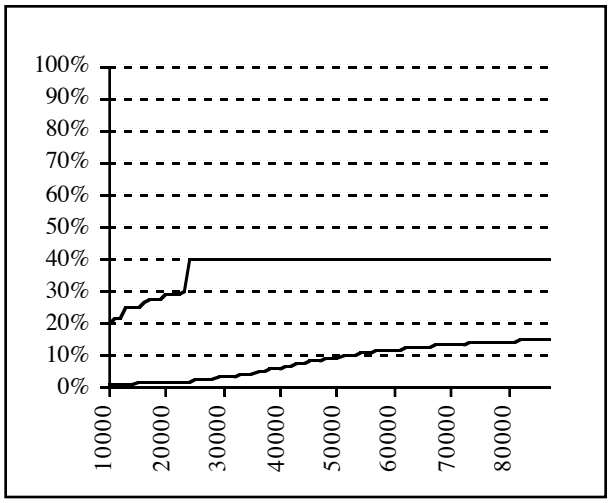

Figure 9: Fitness plots for run z7
(best-of-population and average-of-population).

Acknowledgments

This work was supported by Electronic Arts. The author wishes to thank his supervisor Steve Crane and Vice President of Technology Luc Barthelet, for allowing blue-sky research to coexist with product development. Thanks to John Koza and James Rice for advice. Thanks to "assistant GP guy" Emmanuel Berriet for providing extra CPU cycles. Special thanks to my wife Lisa and to our first child Eric, whose gestation corresponded with this paper's.

References

Altenberg, L. (1994) The Evolution of Evolvability in Genetic Programming, in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA: MIT Press.

Braitenburg, V. (1984) *Vehicles*, MIT press, Cambridge, Massachusetts.

Cliff, D. (1991a) Computational Neuroethology: A Provisional Manifesto, in *From Animals To Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior* (SAB90), Meyer and Wilson editors, MIT Press, Cambridge, Massachusetts.

Cliff, D. (1991b) The Computational Hoverfly; a Study in Computational Neuroethology, in *From Animals To Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior* (SAB90), Meyer and Wilson editors, MIT Press, Cambridge, Massachusetts.

Cliff, D. (1993a) P. Husbands, and I. Harvey Evolving Visually Guided Robots, in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachusetts, pages 374-383.

Cliff, D. (1993b) I. Harvey, and P. Husbands, Explorations in Evolutionary Robotics, *Adaptive Behavior* 2(1), pages 73-110.

Collins, R. J. (1992) *Studies in Artificial Evolution*, Ph.D. thesis, University of California at Los Angeles..

Handley, S. (1994) The Automatic Generation of Plans for a Mobile Robot via Genetic Programming with Automatically defined Functions, in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA: MIT Press.

Harvey, I. (1992) Species Adaptation Genetic Algorithms: The Basis for a Continuing SAGA, in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, Varela and Bourgine editors, MIT Press/Bradford Books, pages 346-354.

Harvey, I. (1993) P. Husbands, and D. Cliff, Issues in Evolutionary Robotics, in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachusetts, pages 364-373.

Kinnear, K. E. Jr. (1993) Generality and Difficulty in Genetic Programming: Evolving a Sort, in *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, Ed San Mateo, CA: Morgan Kaufmann, pages 287-294.

Koza, J. R. (1992) *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, ISBN 0-262-11170-5, MIT Press, Cambridge, Massachusetts.

Koza, J. R. (1994) *Genetic Programming II: Scalable Automatic Programming by Means of Automatically Defined Functions*, MIT Press, Cambridge, Massachusetts (in press).

Mataric, M. J. (1993) Designing Emergent Behaviors: From Local Interactions to Collective Intelligence, in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachusetts, pages 432-441.

Ngo, J. T. (1993) and J. Marks, Spacetime Constraints Revisited, Proceedings of SIGGRAPH 93 (Anaheim, California, August 1-6, 1993), in *Computer Graphics Proceedings*, Annual Conference Series, 1993, ACM SIGGRAPH, New York, pages 343-350.

Reynolds, C. W. (1987) Flocks, Herds, and Schools: A Distributed Behavioral Model, in *Computer Graphics*, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34.

Reynolds, C. W. (1988) Not Bumping Into Things, in the notes for the SIGGRAPH '88 course *Developments in Physically-Based Modeling*, pages G1-G13, published by ACM-SIGGRAPH.

Reynolds, C. W. (1993) An Evolved, Vision-Based Behavioral Model of Coordinated Group Motion, in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachusetts, pages 384-392.

Reynolds, C. W. (1993) An Evolved, Vision-Based Model of Obstacle Avoidance Behavior, in *Artificial Life III*, Santa Fe Institute Studies in the Sciences of Complexity,

Proceedings Volume XVI, C. Langton, Ed. Redwood City, CA: Addison-Wesley.

Reynolds, C. W. (1994a) Evolution of Obstacle Avoidance Behavior: Using Noise to Promote Robust Solutions, in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA: MIT Press.

Reynolds, C. W. (1994b) The Difficulty of Roving Eyes, in *Proceedings of the IEEE World Congress on Computational Intelligence*, IEEE (in press).

Syswerda, G. (1989) Uniform Crossover in Genetic Algorithms, in *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2-9, Morgan Kaufmann Publishers.

Syswerda, G. (1991) A Study of Reproduction in Generational and Steady-State Genetic Algorithms, in *Foundations of Genetic Algorithms*, G. J. E. Rawlins, Ed. San Mateo, CA: Morgan Kaufmann, pages 94-101.

Tackett, W. A. (1994) and A. Carmi, The Donut Problem: Scalability, Generalization, and Breeding Policy in the Genetic Programming, in *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. Cambridge, MA: MIT Press.

von Neumann, J. (1987) Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components, in *Papers of John von Neumann on Computing and Computer Theory*, W. Aspray and A. Burks Eds. Cambridge, MA: MIT Press.

Zapata, R. (1993) P. Lépinay, C. Novales, and P. Deplanques, Reactive Behaviors of Fast Mobile Robots in Unstructured Environments: Sensor-based Control and Neural Networks, in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior* (SAB92), Meyer, Roitblat and Wilson editors, MIT Press, Cambridge, Massachusetts, pages 108-115.

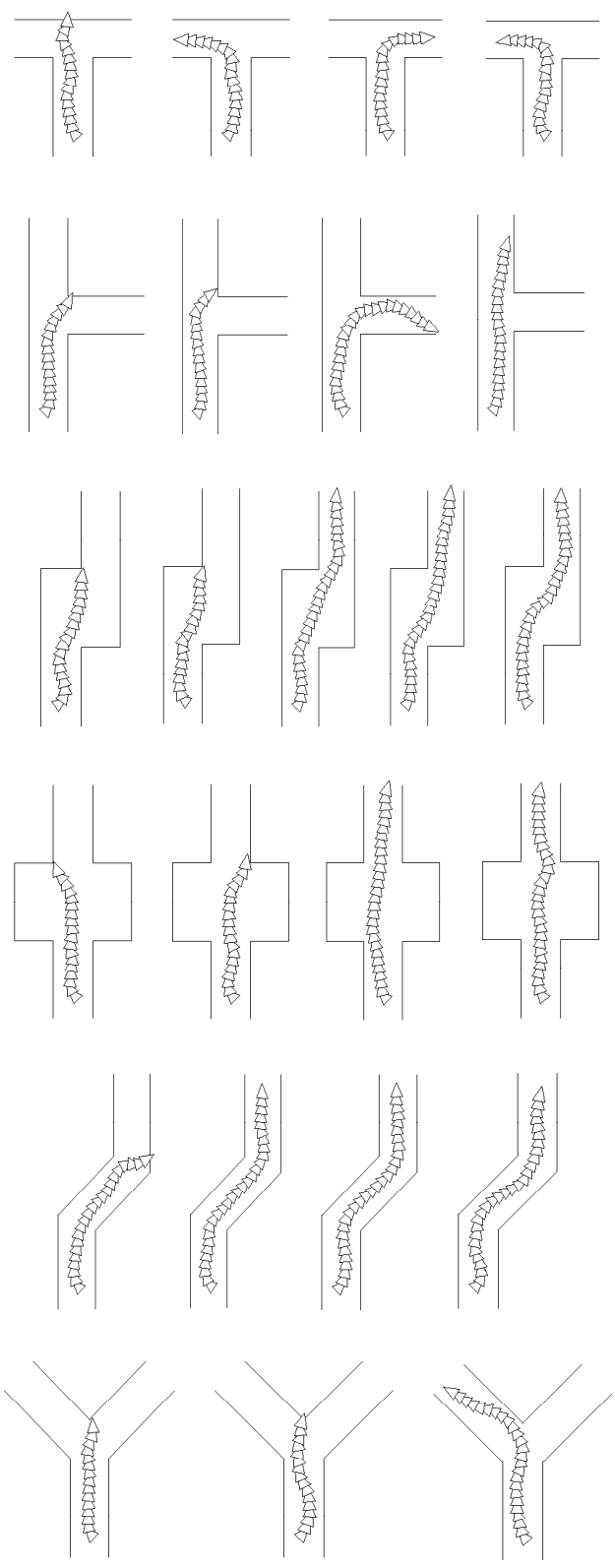

Figure 10: some tests of generalization. Controllers evolved in runs z4 and z6 have been placed in corridors containing novel features. Corridors used in their evolution had only right angles, were of constant width, and contained no branch points.